To modify your `PathFinder` class to find the shortest paths in **directed graphs**, only a few key adjustments are necessary. Currently, the `PathFinder` works for undirected graphs, but directed graphs require handling the directionality of edges. Here's how you can modify the class:

Key Adjustments:

1. **Graph Representation**:
   - Ensure that the graph passed to `PathFinder` is a directed graph. This can be as simple as ensuring that the `Graph<V>` class you're using can support directed edges. Typically, this means the adjacency list `G.adj(v)` will only include vertices that are *reachable* from vertex `v` (i.e., outgoing edges).
2. **Edge Traversal in BFS and DFS**:
   - In undirected graphs, the adjacency list contains both directions of an edge (i.e., for edge `(v, w)`, both `v -> w` and `w -> v` would appear). In directed graphs, the adjacency list only includes outgoing edges, meaning you need no additional modification to the traversal logic. However, you should ensure that your graph client (or the `Graph<V>` class) correctly differentiates between directed and undirected edges when building the adjacency list.
3. **Handling Reverse Paths (Optional)**:
   - If you want to compute shortest paths in both directions in a directed graph (i.e., from `s` to `v` and also from `v` to `s`), you'll need a separate reverse graph that inverts the direction of all edges. This can be useful for algorithms like **Bidirectional Search** or for use cases where reverse paths are meaningful.

Code Changes for Directed Graphs:

Assuming the `Graph<V>` class already handles directed edges, the core change would simply be ensuring that you don't traverse edges in the reverse direction. Here's what that looks like in practice:

1. **BFS Traversal**:
   - The BFS logic doesn't need any change in terms of edge direction, as the adjacency list `G.adj(v)` already contains only the outgoing edges. The BFS will only explore vertices that are reachable via outgoing edges.
2. **DFS Traversal**:
   - The DFS logic similarly needs no change. It will only explore vertices connected by directed edges because the adjacency list contains only outgoing edges.
3. **Optional Reverse Graph**:

- If you want to find shortest paths in the reverse direction (for example, if you also want to find paths *to* the source), you can add support for a reverse graph. This reverse graph would simply swap the direction of all edges.

Modified Code Example for Directed Graphs

Here's an outline of what the code could look like if you also wanted to support the concept of a reverse graph:

```
public class PathFinder<V extends Comparable<V>> {

    private ST<V, V> prev = new ST<>();     // prev[v] = previous
vertex on shortest path from s to v
    private ST<V, Integer> dist = new ST<>(); // dist[v] = length
of shortest path from s to v
    private boolean[] marked;
    private boolean hasPath;
    private Graph<V> graph;                // the directed graph

    // Constructor for BFS shortest path in directed graph
    public PathFinder(Graph<V> G, V s) {
        this.graph = G;
        Queue<V> queue = new Queue<>();
        queue.enqueue(s);
        dist.put(s, 0);  // distance to source is 0

        // Perform BFS, exploring only outgoing edges
        while (!queue.isEmpty()) {
            V v = queue.dequeue();
            for (V w : G.adj(v)) {  // G.adj(v) only returns
outgoing edges
                if (!dist.contains(w)) { // if vertex hasn't been
visited
                    queue.enqueue(w);
                    dist.put(w, 1 + dist.get(v));  // update
distance for directed edge
                    prev.put(w, v);                // track the
previous vertex
                }
            }
        }
    }

    // Optional: Support for finding shortest path in the reverse
direction
    public PathFinder(Graph<V> G, V s, boolean reverse) {
        this.graph = reverse ? G.reverse() : G;
        Queue<V> queue = new Queue<>();
        queue.enqueue(s);
        dist.put(s, 0);  // distance to source is 0
```

```
        // Perform BFS in the reverse graph (if reverse is true)
        while (!queue.isEmpty()) {
            V v = queue.dequeue();
            for (V w : graph.adj(v)) { // G.adj(v) returns outgoing
edges
                if (!dist.contains(w)) {
                    queue.enqueue(w);
                    dist.put(w, 1 + dist.get(v));
                    prev.put(w, v);
                }
            }
        }
    }

    // is v reachable from the source s?
    public boolean hasPathTo(V v) {
        return dist.contains(v);
    }

    // return the length of the shortest path from v to s
    public int distanceTo(V v) {
        if (!hasPathTo(v)) return Integer.MAX_VALUE;
        return dist.get(v);
    }

    // return the shortest path from v to s as an Iterable
    public Iterable<V> pathTo(V v) {
        Stack<V> path = new Stack<>();
        while (v != null && dist.contains(v)) {
            path.push(v);
            v = prev.get(v);
        }
        return path;
    }
}
```

Key Changes:

1. **Directed Graph Traversal**:
   - In both BFS and DFS, we rely on the `Graph<V>` class to only return the outgoing edges. No changes are required for the traversal logic.
2. **Reverse Graph (Optional)**:
   - The constructor includes an optional reverse graph traversal, useful if you want to compute paths in the reverse direction. In this case, you'd need to implement a `reverse()` method in the `Graph<V>` class that creates a new graph with reversed edges.
3. **Graph Class**:

- Make sure that your `Graph<V>` class is aware of edge direction. If it is undirected by default, you'll need to modify how edges are added to ensure that edges are treated as directed.

Summary:

- To find shortest paths in directed graphs, your `PathFinder` class already works without significant changes since BFS and DFS naturally work on directed graphs.
- Ensure that the graph class correctly represents directed edges (i.e., `adj(v)` should return only outgoing edges).
- Optionally, implement a reverse graph if you want to find paths in both directions.

Let me know if you'd like more details on implementing the reverse graph or further enhancements!